V. 15. Бинарная куча - Binary heap. HeapSort.

Общий обзор

Продолжаем и развиваем тему эффективных сортировок. И рассмотрим мы HeapSort. А эта сортировка основана на новой для нас структуре данных - на куче. Да, мы посмотрим только бинарную кучу (Binary Heap), только одного представителя всяких разных и разнообразных куч, никакие другие мы упоминать не будем, но они есть и много. Но и она одна очень приятная штука - у нас появляется не только ещё одна сортировка, но мы ещё и зацепимся за новую для нас структуру данных. Структура, кстати очень любопытная - она по сути своей есть дерево, при этом дерево оказывается настолько специфическим, что его оказывается удобнее хранить в массиве, просто в массиве, не используя никаких явных указателей для организации связей. Более того, дерево - это, кажется, самый рекурсивный сюжет, такое рекурсивное-рекурсивное. Так оказывается, что в данном случае все движения с бинарной кучей реализуются без рекурсии, и притом вполне себе естественно, как-то специально выгибаться, чтобы избавиться от рекурсии хотя бы формально, не надо, всё получается естественно.

И ещё один вопрос: а зачем нам нужна ещё одна сортировка, когда есть уже две, и обе хорошие? Ну, во-первых, есть задачи, в которых HeapSort оказывается лучше, и мы их рассмотрим. А во-вторых, куча, в частности бинарная куча - очень удачная основа для реализации приоритетной очереди (или очереди с приоритетами, priority queue). Ну, это такая очередь, только у всех участников есть приоритеты, и обслуживание происходит в порядке убывания приоритетов, т.е. это не просто FIFO, это существенно более сложная структура, которая прекрасно реализуется на бинарной куче, так что без бинарной кучи мы никуда.

Поехали.

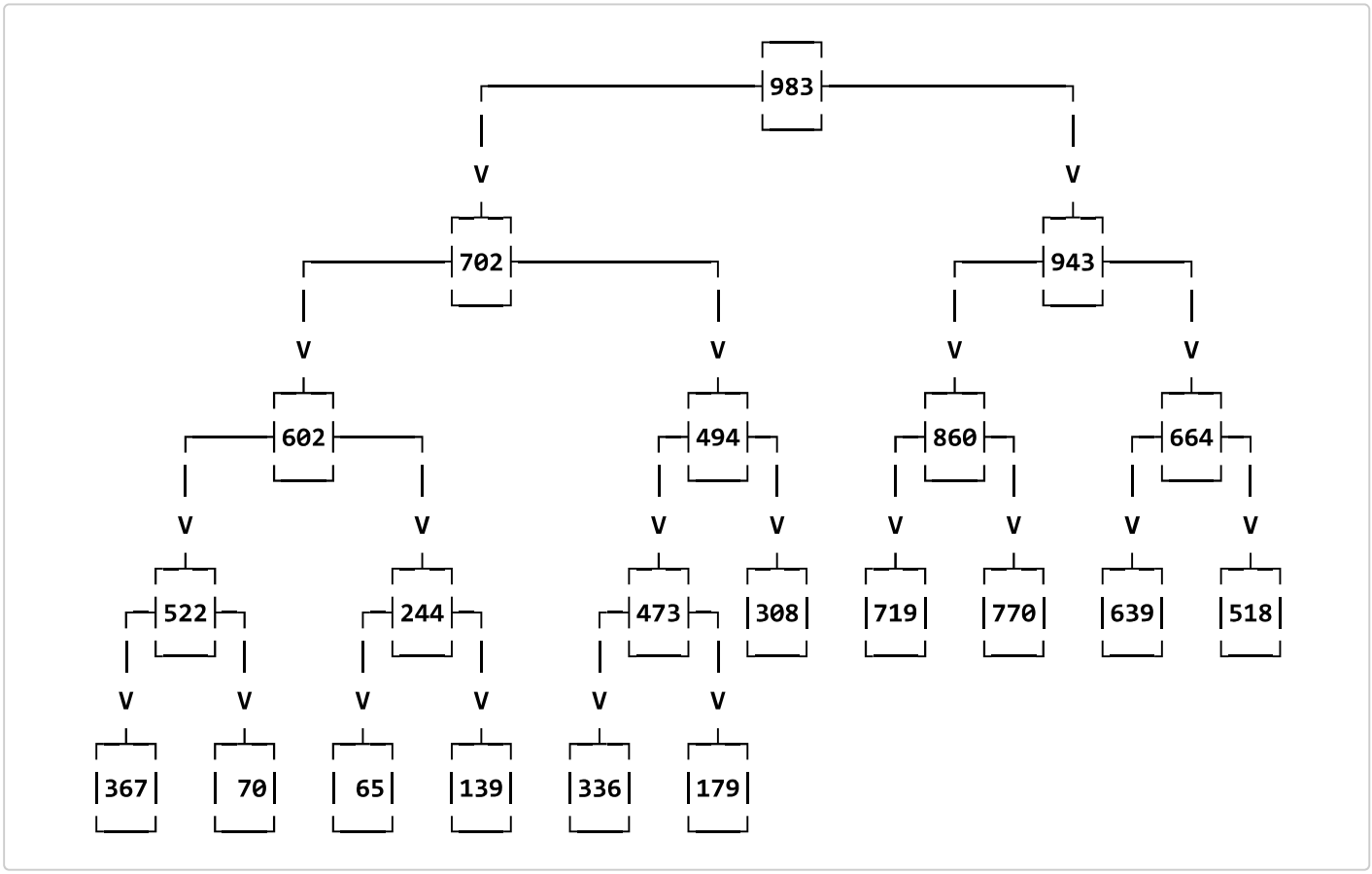
Бинарная куча.

Неформальный взгляд на бинарную кучу.

Бинарная куча - это бинарное дерево, в узлах которого находятся числа. Вообще говоря, не числа, а ключи сортировки, но мы будем, как обычно, считать, что мы сортируем числа - это ничего не меняет. Но у этого бинарного дерева есть некоторые особенности.

1. Первая особенность касается формы бинарного дерева и состоит в том, что в нём нет “дырок”: все ряды бинарного дерева, кроме последнего, заполнены полностью, а последний ряд заполнено слева направо подряд
2. Вторая особенность касается расположения чисел в узлах кучи и состоит в следующем: число в каждом узле дерева, кроме корня, не превосходит числа в родительском узле. Такое свойство дерева называется свойством кучи.

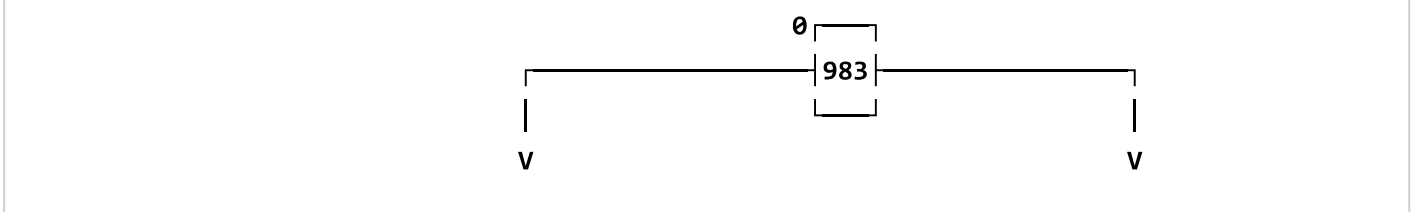
На рисунке приведено бинарное дерево, в котором размещены 21 число: 983, 367, 336, 244, 65, 664, 308, 602, 139, 494, 639, 522, 473, 719, 70, 179, 860, 943, 702, 770, 518.

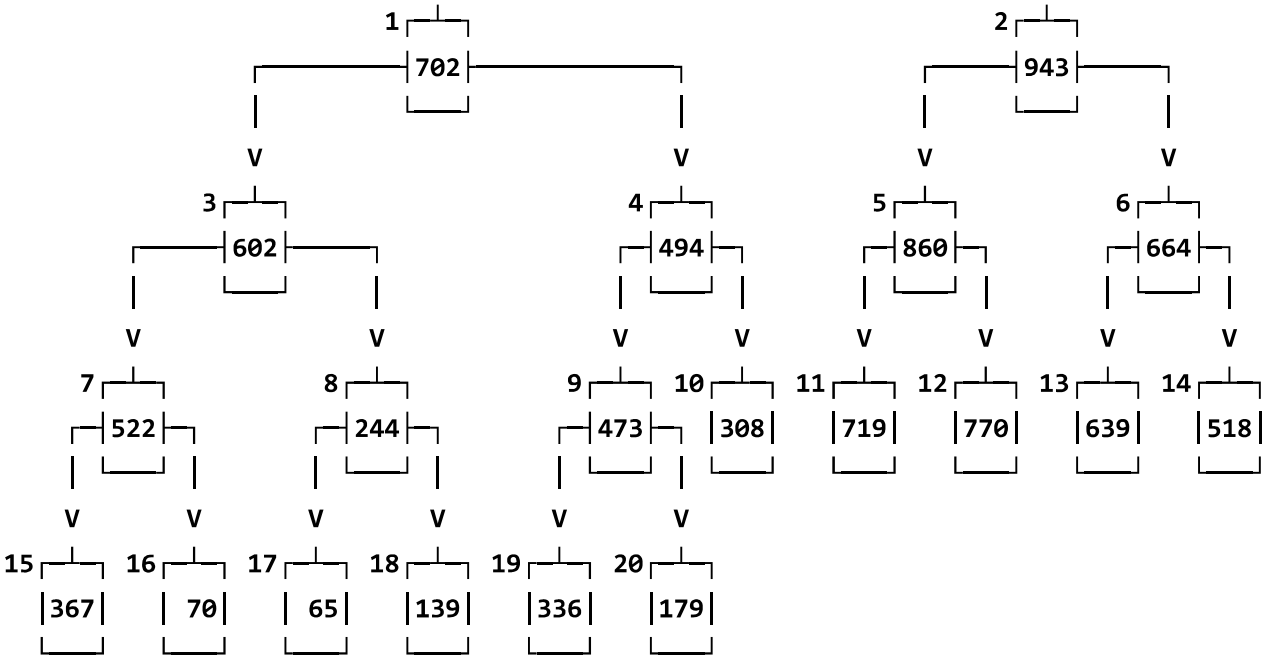


Разумеется, расположить числа в бинарной куче можно разными способами. Вот форма дерева однозначно определяется его размером.

Линеаризация бинарной кучи - хранение кучи в массиве.

Давайте пронумеруем узлы бинарной кучи, двигаясь сверху вниз слева направо, так, как на рисунке:





Ничего особенного в том,чтобы пронумеровать узлы дерева нет, но специфическая форма дерева бинарной кучи обеспечивает замечательное свойство: левый сын узла #K есть узел #2K+1, а правый - узел #2K+2. Соответственно родитель узла #K находится в узле #(K-1)/2, деление целочисленное, т.е. без остатка.

И это замечательное свойство позволяет нам хранить бинарную кучу просто в массиве, не храня никаких указателей на детей/родителей. Именно такую структуру мы и будем рассматривать, мы и будем называть бинарной кучей. Хотя в голове, конечно, будем держать не массив, а именно дерево - думать так много легче. А реализовывать много легче на массиве.

Формальное определение бинарной кучи.

Теперь мы можем формально определить бинарную кучу, а как же можно начинать программировать без формального определения понятий. Особенно в данном случае, когда мы начинаем программировать новую структуру данных.

Определение-1. Отрезок массива a[L:R] (в него входят R-L элементов от a[L] до a[R-1] включительно) обладает свойством кучи, если для всех элементов этого отрезка a[K] (L<=K<R) выполняются условия:

```
- если 2*K+1 < R, то a[K] >= a[2*K+1]
- если 2*K+2 < R, то a[K] >= a[2*K+2]
```

Определение-2. Отрезок массива a[L:R] (в него входят R-L элементов от a[L] до a{R-1] включительно) обладает свойством кучи, если для всех элементов этого отрезка a[K] (L<=K<R)

выполняется условие:

```
- если (K-1)/2 >= L, то a[K] <= a[(K-1)/2]
```

Эквивалентность двух опередедений очевидна. Первое утверждает, что каждый элемент не меньше своих сыновей, второе - что каждый элемент не превосходит своего родителя.

Маленькое методическое указание. Очень запутанное, но желательно с ним разобраться. Я здесь умышленно не использую слово слайс, чтобы подчеркнуть, что речь идёт именно о фрагменте целого массива, а не об отдельной переменной, об отдельной структуре. Впрочем, этот массив вполне себе может оказаться при реализации слайсом, но тут имеется в виду именно цельная переменная, цельная структура. Кажется я всё запутал, остановлюсь на всякий случай. Да, и я использую слайсовые обозначения - уж очень они естественны, никуда от них не денешься.

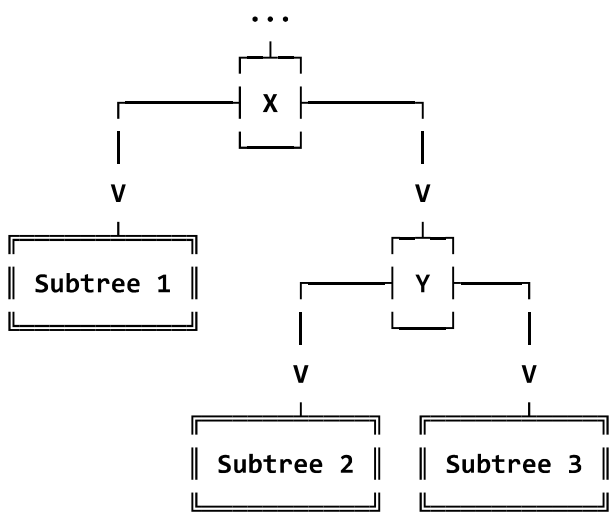
Определение. Бинарная куча - это массив, обладающий свойством кучи.

Очевидное свойство кучи. В корне располагается наибольший элемент кучи.

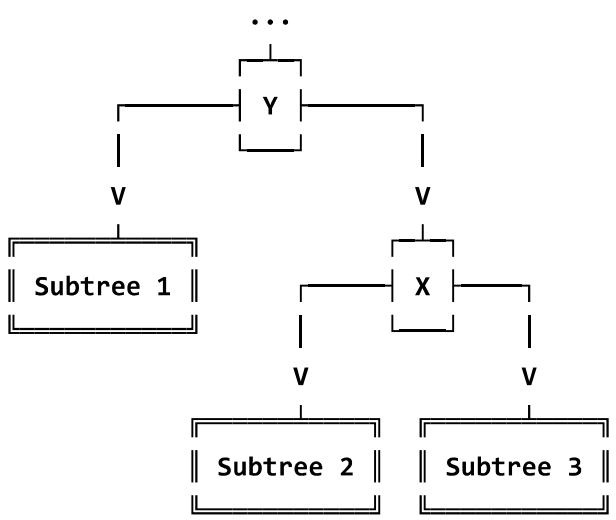
Бинарная куча. Базовые операции.

Подъём.

Допустим, что у нас есть куча, и какой-то элемент увеличился. На рисунке изображено поддерево элемента X (X не обязательно лежит в корне всего дерева), а увеличился элемент Y.



В результате, если Y > X, то нарушается свойство кучи. Поменяем X и Y местами.



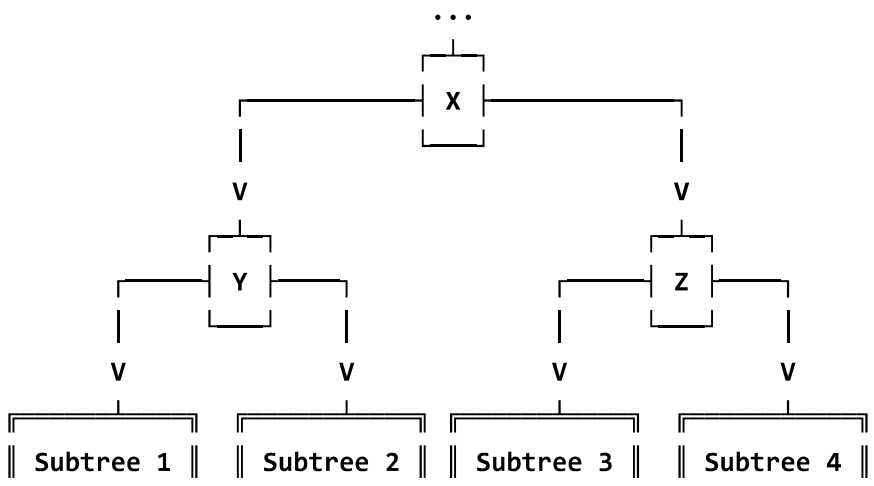
В результате поддерево, изображённое на рисунке, обладает свойством кучи. В самом деле до начала всех движений X был наибольшим в изображённом дереве, так что и теперь все элементы поддеревьев Subtree2 и Subtree3 не превосходят X. С другой стороны Y > X, так что он точно больше всех остальных элементов.

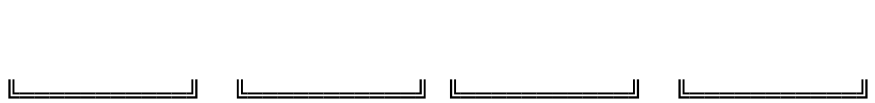
Снова сравниваем Y со своим родителем и при необходимости меняем их местами. Повторяем процесс до тех пор, пока не окажется, что Y не превосходит своего родителя, либо мы не поднимем его в корень всего дерева.

При реализации, конечно, не будем менять местами элементы, а просто X записывать на место Y. Значение же Y будем записывать только один раз - в конце подъёма.

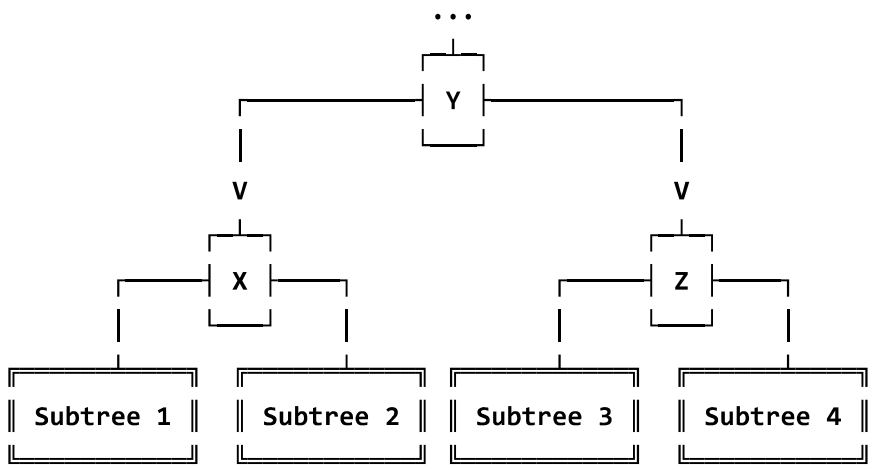
Спуск.

Допустим, что у нас есть куча, и какой-то элемент уменьшился. На рисунке уменьшается элемент X (X не обязательно лежит в корне всего дерева), и изображено его поддерево. В результате X может стать меньше одного или обоих своих сыновей, и свойство кучи нарушится.





Сравним сыновей X. Если Y >= Z, то меняем местами X и Y. В результате в корне изображённого поддерева лежит его наибольший элемент: Y > X, Y >= Z, а Z >= всех элементов в обоих своих поддеревьях.



Остаётся восстановить возможно нарушенное свойство кучи для поддерева, в корне которого лежит X. Для этого продолжаем опускать X до тех пор, пока он не станет больше или равен всех своих сыновей или сыновей не останется - дойдём до листа.

Если Y < Z, то действуем аналогично.

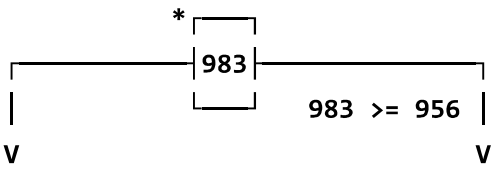
Как и при подъёме, в реализации не будем менять местами элементы, а просто Y записывать на место X. Значение же X будем записывать только один раз - в конце спуска.

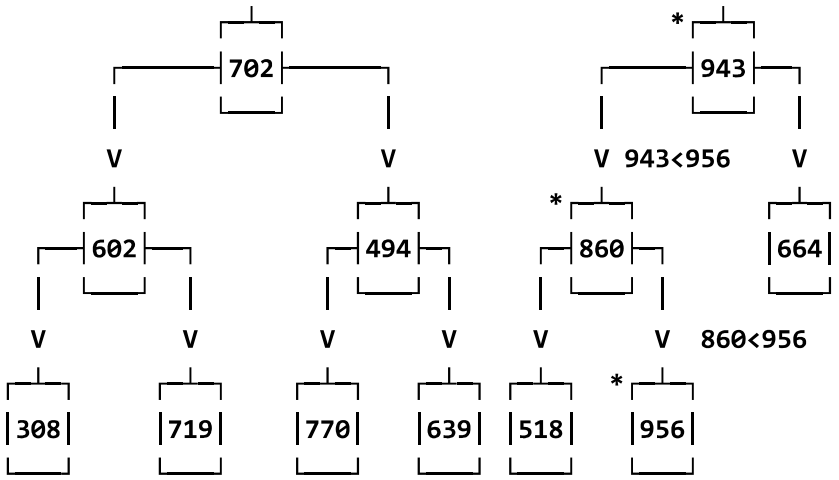
Бинарная куча. Основные операции.

Добавление элемента.

Увеличиваем размер кучи на 1 и добавляем новый элемент в конец массива, после чего совершаем подъём этого элемента. Можем считать, что мы добавили в конец кучи минус бесконечность, а потом увеличили последний элемент кучи.

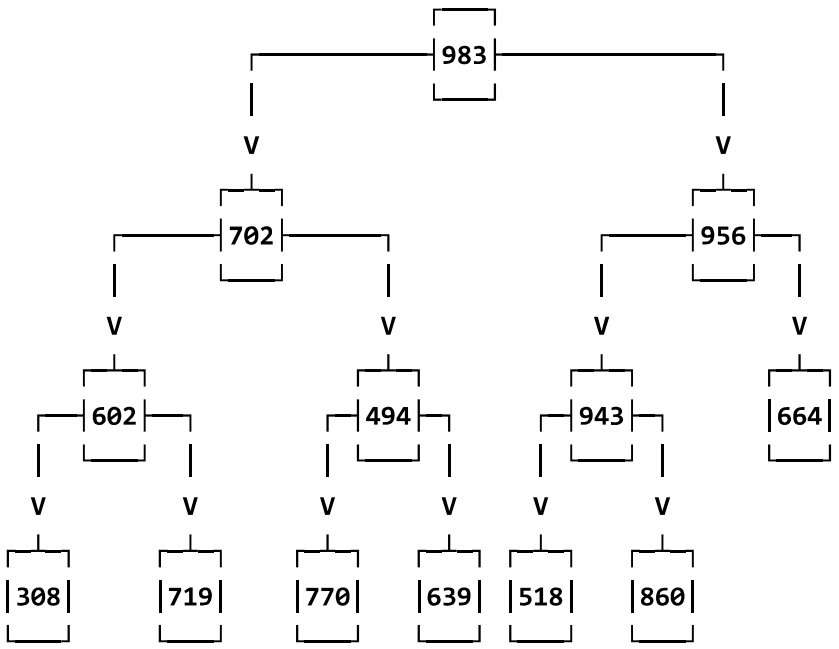
На рисунке изображён процесс добавления нового элемента - 956 - в бинарную кучу.
Исходная куча.





Звёздочками отмечены элементы кучи, участвующие в процессе подъёма.

В итоге получаем:



Удаление корневого элемента.

Ставим последний элемент на место корневого элемента (на место #0) и уменьшаем размер кучи на 1. После этого опускаем нулевой элемент.

Изменение элемента.

Изменяем элемент кучи. Если он при этом уменьшился, то опускаем его, если увеличился - поднимаем.

Удаление произвольного элемента.

Заменяем удаляемый элемент на последний элемент кучи и уменьшаем размер кучи на 1. Затем действуем так, как если бы мы изменили удаляемый элемент (а мы именно это и сделали). Замечу, что последний элемент может оказаться больше удаляемого.

Превращение произвольного массива в кучу - окучивание, или хипофизация.

Казалось бы, у нас всё уже есть: начинаем с пустой кучи и добавляем элементы по одному. Однако оказывается существенно быстрее заполнять кучу с конца: ставить элемент на последнее место, потом на предпоследнее и т.д. вплоть до нулевого, следя каждый раз, чтобы отрезок массива от K-го (K - это номер позиции, на которую ставим очередной элемент) до последнего обладал свойством кучи. Почему получается выигрыш в скорости? Сразу видно, что вторая половина массива вообще не требует никаких действий по “окучиванию” отрезка. Затем (примерно четверть элементов) может опуститься (а каждый новый элемент может только опускаться) максимум на один шаг, примерно одна восьмая часть всего массива может опуститься максимум на два шага и т.д. Т.е. процедура спуска исполняется быстрее - если вставлять новый элемент в конец кучи, то примерно половина элементов может подниматься на всю высоту кучи и т.д. Это очевидно требует больше операций. Более точно, но это уже несколько мелкий шрифт, об этом говорить, видимо не стОит, предложенный вариант хипофизации имеет сложность O(N), а добавление элементов в конец кучи имеет сложность O(N*logN), где N - размер кучи.

Программная реализация.

файл binheap.go

```
package binheap

import (
    "math"
)

type (
    BinaryHeap []Tdata
    Tdata int32
)

const (
    MinData = math.MinInt32
)
```

```go
func (b *BinaryHeap) Init (data []Tdata) {
    for _, x := range data {
        (*b) = append(*b, x)
    }
    (*b).Heapify()
}

func (b BinaryHeap) pushUp(place int) {
    if place >= len(b) || place <= 0 {
        return
    }
    x := b[place]
    parent:= (place-1)/2
    for  place > 0 && b[parent] < x {
        b[place] = b[parent]
        place = parent
        parent = (place-1)/2
    }
    b[place] = x
}

func (b BinaryHeap) pushDown(place int) {
    if place >= len(b) || place < 0 {
        return
    }
    x := b[place]
    for  {
        if 2*place + 1 >= len(b) { // лист - сыновей нет
            break
        }
        maxson := 2*place + 1  // левый сын
        rson:= maxson + 1
        if rson < len(b) && b[rson] > b[maxson] { // правый сын больше левого
            maxson = rson
        }
        if b[maxson] <= x {
            break
        }
        b[place] = b[maxson]
        place = maxson
    }
    b[place] = x
}
```

```go
func (b *BinaryHeap) Add(value Tdata) {
    (*b) = append(*b, value)
    (*b).pushUp(len(*b)-1)
}

func (b BinaryHeap) GetMax() Tdata {
    if len(b) > 0 {
        return b[0]
    } else {
        return MinData
    }
}

func (b *BinaryHeap) ExtractMax() Tdata {
    if len(*b) > 0 {
        max:= (*b)[0]
        (*b)[0] = (*b)[len(*b)-1]
        *b = (*b)[:len(*b)-1]
        (*b).pushDown(0)
        return max
    } else {
        return MinData
    }
}

func (b *BinaryHeap) Delete(place int) {
    if place >= len(*b) {
        return
    }
    x:= (*b)[len(*b)-1]
    *b = (*b)[:len(*b)-1]
    (*b).Change(place, x)
}

func (b BinaryHeap) Change(place int, value Tdata) {
    b[place] = value
    if place > 0 && value > b[(place - 1) / 2] {
        b.pushUp(place)
    } else {
        b.pushDown(place)
    }
}
```

```
func (b BinaryHeap) Heapify() {
    for k:= (len(b) / 2) - 1; k >= 0; k-- {
        b.pushDown(k)
    }
}
```

Создаём каталог `C:\Go\src\binheap` , закидываем в него этот исходный текст в файле binheap.go, создаём
программу heapsort.go:

```
package main

import (
    "fmt"
    "binheap"
)

func HeapSort(a []binheap.Tdata) {
    var bheap binheap.BinaryHeap
    bheap.Init(a)
    for k:= len(a) - 1; k >= 0; k-- {
        a[k] = bheap.ExtractMax()
    }
}

func main() {
    a:= []binheap.Tdata{2,5,7,2,4,9,1,6}
    fmt.Println(a)
    HeapSort(a)
    fmt.Println(a)
}
```

компилируем, запускаем и - да, всё работает.

И всё-таки, вернёмся к вопросу, который прозвучал в самом начале:

а зачем нам нужна ещё одна сортировка, когда есть уже две - QuickSort и MergeSort, - и обе хорошие?

Самое главное отличие HeapSort от двух других сортировок состит в том, что HeapSort выдаёт числа по одному, т.е. он, с одной стороны, выдаёт результаты в реальном времени, с

приблизительно одинаковыми интервалами, а, с другой стороны, он работает примерно также быстро (ну, качественных отличий по скорости нет), как и два других алгоритма. В этом смысле он действует примерно также, как и сортировка простым выбором (пробегаем по массиву и выбираем наибольшее/наименьшее число), но, в отличие от сортировки простым выбором, он работает быстро. QuickSort и MergeSort окончательных результатов по ходу выполнения не дают (ну, или мало что дают), чтобы получить результат, надо дожидаться окончания сортировки всего массива. И это его свойство оказывается очень ценным во многих задачах. Перечислю некоторые характерные применения HeapSort - и какие-то общие подходы, и некоторые конкретные задачи.

1. Уже упоминавшиеся во вступлении приоритетные очереди. Очереди, в которых обслуживание происходит в порядке убывания приоритетов. При этом в очередь могут приходить новые элементы, а обслуженные - уходят.
2. Вообще, все задачи, в которых множество элементов динамически изменяется, а нам надо также динамически находить максимальный/минимальный (или первые несколько максимальных/минимальных) элементы. В частности, куча используется при реализации алгоритма Дейкстры для нахождения кратчайшего пути в графе или алгоритма Прима для построения минимального стягивающего дерева в графе. Но ограничусь только названиями.
3. Найти K наибольших/наименьших элемента массива. Если K заметно меньше длины массива, то HeapSort - очень хорошее решение.
4. Давайте рассмотрим такую модификацию MergeSort: делим массив не на 2, а на K частей, сортируем их рекурсивно, а потом эти части склеиваем. При этом нам как раз надо выбирать наименьшее из K чисел, забрасывать его в результирующий массив, а потом заменять на другое - следующее в той части, в которой жил выбранный элемент. На этапе склейки HeapSort очень даже уместен.

Задание. Написать MergeSort с разделением на K частей, а при склейке изспользовать HeapSort. Побенчмарчить для различных значений K.

Микрозадание до сих пор речь шла о max-куче - куче, в которой значение в родителе больше или равно значениям в сыновьях. Измените max-кучу на min-кучу - кучу, в которой значение в родителе меньше или равно значений в сыновьях.

5. Назовём массив почти отсортированным степени K, если в нём каждый элемент отстоит от своей позиции в отсортированном массиве не более, чем на K позиций.

Пусть у нас есть почти отсортированный массив степени K. Тогда его можно отсортировать таким образом. Берём первые K+1 число массива, строим из них min-кучу, выбираем их них наименьшее, извлекаем его из кучи, а вместо него в кучу вставляем следующее число из массива. Тоже интересное применение, особенно если вспомнить про вариант Сэджвика в QuickSort'e.

Задание В варианте Сэджвика в QuickSort'e в конце вместо одного прохода сортировками простыми обменами, реализовать вышеописанный способ. Побенчмарчить для различных значений cutoff. Напомню, что в данном случае, cutoff - это тот размер, при котором (и меньшем)

отрезок массива уже не сортируется рекурсивно, а остаётся на один проход окончательной сортировки.

Локаторы.

А что делать, если у нас удаляется из кучи не максимальный/минимальный элемент, который понятно, где находится, а удаляется или изменяется какой-то конкретный элемент? Нет, как это сделать, мы знаем, а вот как его найти, где он находится в куче? Иначе говоря, задача такова: все элементы исходного массива пронумерованы. Мы строим из них кучу - переставляя при этом элементы, затем как-то её модифицируем. Требуется в любой момент времени указать, на каком месте в куче находится элемент #K.

Для решения этой проблемы заведём для каждого элемента локатор - переменную, в которой постоянно хранится текущее положение соответствующего элемента в куче. Определяем соответствующие типы:

```
type (
    Lmnt struct {
        Index int
        Value Tdata
    }
    LocatorBinaryHeap struct {
        heap []Lmnt
        locator []int
    }
)
```

Общая идея такова. Инициализируем кучу массивом (слайсом) с данными. Каждый элемент массива (будем для простоты называть его массивом, подразумевая, что он может быть слайсом) данных закидываем в кучу с локатором посредством

```
func (b *LocatorBinaryHeap) Init (data []Tdata)
```

В результате b.heap содержит все данные, а b.locator[index] содержит место элемента {Index, Value} (соответствующего data[index]) в куче b.heap. А дальше мы можем исполнять с кучей различные операции.

- Возвращает наибольший элемент данных

  ```
  func (b LocatorBinaryHeap) GetMax() Lmnt
  ```

- Удаляет из кучи элемент с заданным индексом. Если этот элемент сейчас не в куче, то ничего не происходит (вообще-то надо было бы выдавать ошибку, но не будем слишком уж загружать)

```
func (b *LocatorBinaryHeap) Delete(index int)
```

- Возвращает в кучу элемент. Если элемент находится в куче, то блокируем такую попытку (тут тоже надо было бы генерить ошибку, но не будем…). При этом можно изменить значение Value. Конечно, если это важно по ходу пьесы, надо бы соответствующим образом изменять элементы того массива, которым мы инициализировали кучу.

```
func (b *LocatorBinaryHeap) Add(t Lmnt)
```

- Изменяет значение элемента в куче. Если элемент не находится в куче, то блокируем такую попытку (и тут тоже надо бы генерить ошибку, но тоже не будем…).

```
func (b LocatorBinaryHeap) Change(t Lmnt)
```

- Возвращает корневой (максимальный) элемент кучи и удаляет его из кучи.

```
func (b *LocatorBinaryHeap) ExtractMax() Lmnt
```

  В этой реализации есть одна очень важная особенность: **размер локатора не изменяется никогда, кроме как при нициализации**. Т.е. мы не должны изменять размер массива данных. Данные могут изменяться, каждый элемент может входить в кучу или не входить, но изменять объём данных или пытаться воткнуть в кучу элемент с отрицательным индексом или индексом, превосходящим размеры локатора (а его длина устанавливается при нициализации и потом не меняется) запрещено. Такой вариант видится довольно естественным, но если мы захотим снять его ограничения, добавлять элементы с новыми индексами, с индексами, которых не было в момент инициализации, то надо хранить локатор в мапе. Кучу, конечно, храним в слайсе - ведь куча, мы это уже поняли выше, - это массив (слайс). Но не буду здесь расписывать подробности. Если кого-то из детей это заинтересует настолько, что они захотят **модифицировать кучу с локатором (прекрасное задание, кстати)**, разберутся в деталях. Упражнение очень сильное. Чтобы его выполнить, надо осознать что-то, погрузиться на некоторую глубину. В то же время задание технически не очень сложное, так что было бы интересно…

Исходный код бинарной кучи с локатором ставить сюда, видимо, особо смысла нет - всё, включая версию для печати в pdf, есть в материалах к этому занятию.

Задачи для самостоятельной работы

Одно задание - реализовать бинарную кучу с локатором, хранящимся в мапе, уже сформулировано. А ниже я приведу пару задач на применение бинарной кучи.

Пару задач немного нетрадиционных. Они, такие, олимпиадные. Там до бинарной кучи ещё добраться надо, хотя она всё решает. Но имеются довольно подробные спойлеры к этим задачам. И бинарная куча играет в решениях основную роль. Да, и ещё к задачам имеются комплект тестов, так что проверять решения тоже довольно удобно.

Задачи, тесты и прочие материалы к ним собраны в каталоге `Tasks` .

1. Задача “Электрички”. Просто строим две кучи. Я не буду здесь расписывать задачу, текст задачи и анализ есть.
2. “Последняя” задача. Там тоже всё решает куча, но дорога к ней довольно длинная и неочевидная. Но анализ задачи приведён очень подробный, так что лучше его почитать. Но у этой задачи есть второй, и очень глубокий, слой. Эта задача - это довольно удачно замаскированный алгоритм Хаффмана для построения оптимального префиксного кода. Тема очень интересная, но это уже совсем про другое. Кого это заинтересует, в приложении к этой задаче лежат соответствующие фрагменты из нескольких книг, книги одна другой лучше, но остановимся на этом…